UNIT 2
PERFORMANCE

Compiled By
Ms. Shruthi M
Dept of CSE.

What is performance testing / engineering???

- Act of validating an application and its
- deployed environment including hardware
- / network against performance goals like
- Responsiveness, Reliability, Scalability
- with the intent of optimizing
- the system behaviour to the desired level.

Introduction

- Long ago – Efforts were made to make program efficient coz computers were slow and expensive.
- Present day – Machines are cheaper and faster.

- Why should we worry about performance:
- When the problem is important –
- 1. but it is slow and it meets expectations like correctness, robustness and clarity…
- 2. Or program is fast – but wrong answer, so time also is lost.

- Programs written as assignments, personal programs are rarely used later – speed not so important.

- Runtime of a commercial product, central component such as a graphics library – critically important tasks, performance important.

- Speed , Space are important concerns to measure performance.

- Best strategy : Use the simplest, cleanest algorithms and data structures appropriate for the task.

- Then measure performance to see if changes are needed: Enable compiler options to generate fastest possible code, assess what changes to program itself will have most effect, make changes one at a time, re-assess, keep simple versions for testing revisions.

- Measurement – crucial component for performance improvement. Use tools like timing commands and profilers.

Performance: A Bottleneck

- Scenario: How bottleneck was removed from a critical program

- Our incoming mail funnels through a machine called a gateway, that connects our internal network with the external Internet.
- Electronic mail messages from outside - tens of thousands a day for a community of a few thousand people - arrive at the gateway and are transferred to the internal network; this separation isolates our private network from the public Internet and allows us to publish a single machine name (that of the gateway) for everyone in the community.

- One of the services of the gateway is to filter out "spam," unsolicited mail that advertises services of dubious merit.
- After successful early trials of the spam filter, the service was installed as a permanent feature for all users of the mail gateway, and a problem immediately became apparent.
- The gateway machine, antiquated and already very busy, was overwhelmed because the filtering program was taking so much time-much more time than was required for all the other processing of each message-that the mail queues filled and message delivery was delayed by hours while the system struggled to catch up.

- This scenario is an example of a true performance problem: the program was not fast enough to do its job and people were inconvenienced by the delay.

- Simplifying quite a bit, the spam filter runs like this.
- Each incoming message is treated as a single string, and a textual pattern matcher examines that string to see if it contains any phrases from known spam, such as
- "Make millions in your spare time“ or "XXX-rated."

- Messages tend to recur, so this technique is remarkably effective, and if a spam message is not caught, a phrase is added to the list to catch it next time.

- None of the existing string-matching tools, such as grep, had the right combination of performance and packaging. so a special-purpose spam filter was written.

- The original code was very simple; it looked to see if each message contained any of the phrases (patterns):

```
/* isspam: test mesg for occurrence of any pat */
    int isspam(char *mesg)
    {
        int i;
        for (i = 0;  i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL) {
                printf ("spam: match for '%s'\n”, pat [i]) ;
                return 1;
        }
        return 0;
     }
```

- How could this be made faster?

- The string must be searched, and the s t r s t r function from the C library is the best way to search: it's standard and efficient.
- By changing the way strstr worked, it could be made more efficient for this particular problem
- The existing implementation of strstr looked something like this:
- /* simple strstr : use strchr to look for first character */
- char *strstr(const char *s1, const char *s2)
- {
- int n;
- n = strlen(s2);
- for (;;) {
- sl = strchr(s1, s2[0]);
- if (sl == NULL)
- return NULL;
- if (strncmp(s1, s2, n) == 0)
- return (char *) s1 ;
- sl++ ;
- }
- }

- For typical use it was fast because it used highly-optimized library routines to do the work.
- It called strchr to find the next occurrence of the first character of the pattern, and then called strncmp to see if the rest of the string matched the rest of the pattern. Thus it skipped quickly over most of the message looking for the first character of the pattern. and then did a fast scan to check the rest.

Why would this perform badly?

- First, strncmp takes as an argument the length of the pattern. which must be computed with strlen. But the patterns are fixed, so it shouldn't be necessary to recompute their lengths for each message.
- Second, strncmp has a complex inner loop. It must not only compare the bytes of the two strings, it must look for the terminating \0 byte on both strings while also counting down the length parameter. Since the lengths of all the strings are known in advance (though not to strncmp), this complexity is unnecessary; we know the counts are right so checking for the \0 wastes time.

- Third, strchr is also complex, since it must look for the character and also watch for the \0 byte that terminates the message. For a given call to isspam, the message is fixed, so time spent looking for the \0 is wasted since we know where the message ends.
- Finally, although strncmp, strchr, and strlen are all efficient in isolation, the overhead of calling these functions is comparable to the cost of the calculation they will perform.
- It's more efficient to do all the work in a special, carefully written version of strstr and avoid calling other functions altogether.
- The existing strstr was fine when both the pattern and the string were short and changed each call, but when the string is long and fixed, the overhead is prohibitive.
- strstr was rewritten to walk the pattern and message strings together looking for matches, without calling subroutines. The resulting implementation has predictable behavior: it is slightly slower in some cases, but much faster in the spam filter and, most important, is never terrible.

- To verify the new implementation's correctness and performance, a performance test suite was built.

- Test suite:
- Looking for a pattern of a single x in a string of a thousand e's and a pattern of a thousand x's in a string of a single e, both of which can be handled badly by naive implementations.
- Such extreme cases are a key part of performance evaluation.
- The library was updated with the new strstr and the spam filter ran about 30% faster, a good payoff for rewriting a single routine.
- Unfortunately, it was still too slow.
- The real problem is to search for a large, fixed set of textual patterns in a long, variable string.
- Put that way, strstr is not so obviously the right solution.
- The most effective way to make a program faster is to use a better algorithm.

- With a clearer idea of the problem, it's time to think about what algorithm would work best.
- The basic loop,

```
for (i = 0; i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL)
return 1;
```

- scans down the message npat independent times; assuming it doesn't find any matches, it examines each byte of the message npat times, for a total of
- strlen (mesg) *npat comparisons.
- A better approach is to invert the loops, scanning the message once in the outer loop while searching for all the patterns in parallel in the inner loop:

```
for (j = 0; mesg[j] != '\0'; j++)
        if (some pattern matches starting at mesg[jl)
                return 1;
```

- Performance improvement:
- To see if any pattern matches the message at position j, we don't need to look at all patterns, only those that begin with the same character as mesg[j].
- Roughly, with 52 upper and lower-case letters we might expect to do only strlen(mesg)*npat/52 comparisons.

- Since the letters are not evenly distributed-words begin with s much more often than x-we won't see a factor of 52 improvement, but we should see some.
- In effect, we construct a hash table using the first character of the pattern as the key.

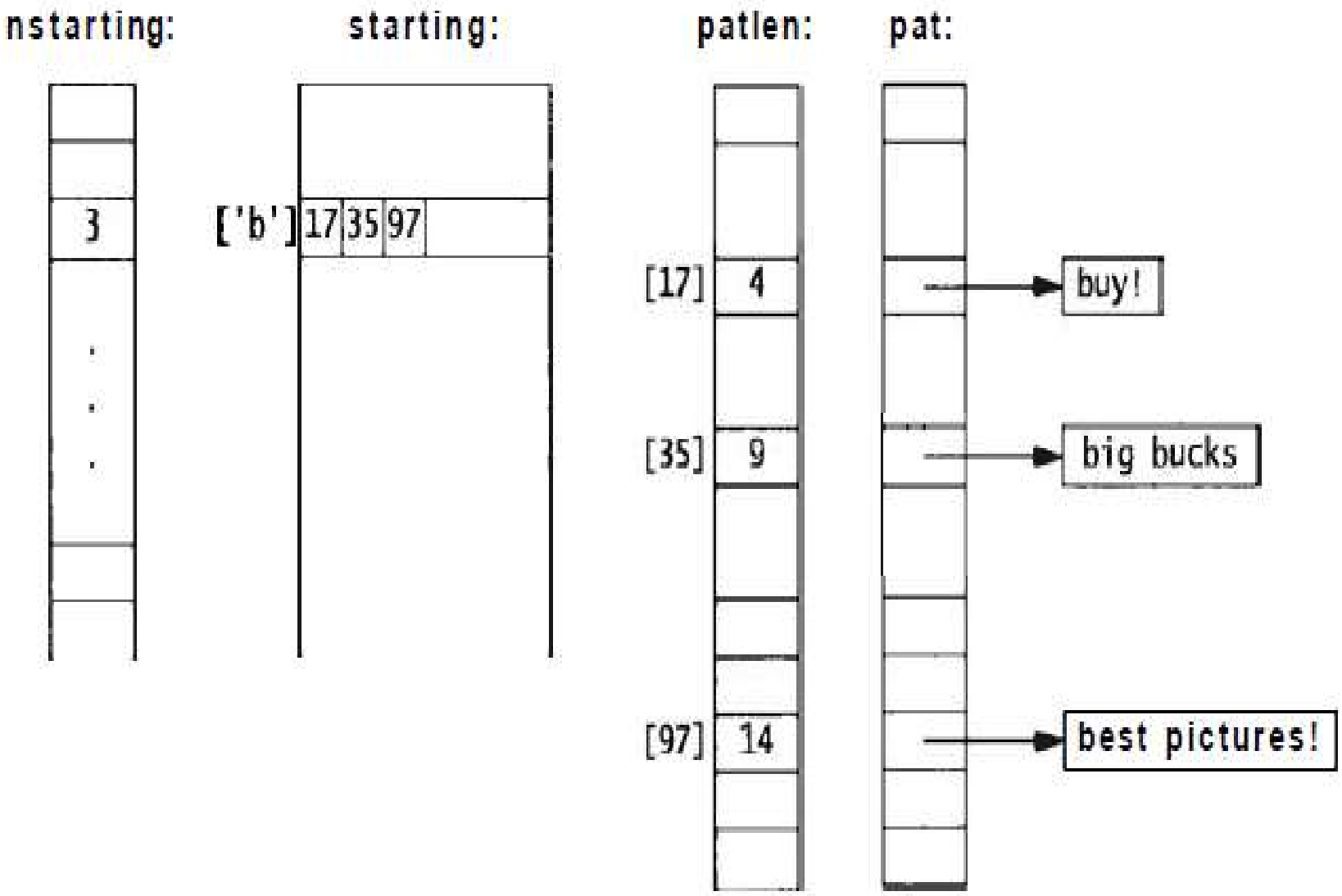
- isspam is still short, when given some precomputation to construct a table of which patterns begin with each character:

```
int patlen[NPAT];                          /* length of pattern */
int starting[UCHAR_MAX+1][NSTART]; /* pats starting with char */
int nstarting[UCHAR_MAX+1];          /* number of such patterns */
...
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i, j, k;
    unsigned char c;

    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i= 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k] , patlen[k]) == 0) {
                printf("spam: match for '%s'\n", pat[k]);
                return 1;
            }
        }
    }
    return 0;
}
```

- The two-dimensional array starting [c] [ ]stores, for each character c, the indices of those patterns that begin with that character.
- Its companion nstarting[c] records how many patterns begin with c.

- Without these tables, the inner loop would run from 0 to npat, about a thousand; instead it runs from 0 to something like 20.
- Finally, the array element patlen[k] stores the precomputed result of
- strlen(pat [k]).


nstarting:
starting:
patlen:
pat:
3
['b']
17
35
97
[17]
4
[35]
9
[97]
14
buy!
big bucks
best pictures!

- The code to build these tables is easy:
- int i;
- unsigned char c;
- for (i =0; i < npat; i++) {
- c = pat[il[o];
- i f (nstarting [c] >= NSTART)
- eprintf ("too many patterns (>=%d) begin '%c"' , NSTART, c);
- starting[c] [nstarting[c]++] = i ;
- patlen[i] = strlen(pat[i]);
- }
- Depending on the input, the spam filter is now five to ten times faster than it was using the improved strstr , and seven to fifteen times faster than the original implementation. Performance problem solved.